苏州大学实验报告

院、系	计算机学院	年级专业	软件工程	姓名	朱金涛	学号	2327406014
课程名称	操作系统课程实践					成绩	
指导教师	王红玲	同组实验者	无		实验日期	2025 年 10 月 20 日	

实 验 名 称 内核模块实验

一.实验目的

（1）掌握利用内核模块机制实现较复杂功能的办法。
（2）学习并掌握/proc 文件系统的创建。

二.实验内容

（1）编写一个简单的具备基本要素的内核模块，其功能要求仅在控制台输出“Hello World!”，并编写这个内核模块所需要的 Makefile，最后编译内核并将其载入系统。
（2）利用内核模块在/proc 模块下创建 proc_example 目录，并在该目录下创建三个普通文件(foo, bar, jiffies)和一个链接文件(jiffies_too)。

三.实验步骤和结果

1. 整体目录结构

```
kernel_module_lab/
│
├── helloworld.c          ← 实验一模块源码
├── Makefile              ← 对应 helloworld.c 的 Makefile
│
├── procfs_example.c      ← 实验二模块源码
└── Makefile_proc         ← 对应 procfs_example.c 的 Makefile
```

2. 具体实现

（1）简单内核模块（Helloworld）

Helloworld.c：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");                                          // 模块许可
MODULE_DESCRIPTION("A simple hello world kernel module"); // 模块功能描述
MODULE_AUTHOR("Jintao");                                        // 作者信息（可选）
MODULE_VERSION("1.0");                                          // 版本号（可选）

// 初始化函数
static int __init hello_init(void)
{
```

```
    printk("<1>Hello, World!\n"); // 打印日志
    return 0;                     // 成功加载
}

// 清理函数
static void __exit hello_exit(void)
{
    printk("<1>Goodbye, World!\n"); // 打印退出信息
}

module_init(hello_init); // 注册初始化函数
module_exit(hello_exit); // 注册清理函数
```

本 C 文件编写了一个最简单的 Linux 内核模块 helloworld，通过实现模块的加载与卸载过程，初步理解了内核模块的结构与工作机制。代码中包含两个核心函数：hello_init() 和 hello_exit()，分别在模块加载与卸载时执行。加载模块后，printk() 函数向内核日志输出“Hello, World!”，卸载模块时输出“Goodbye, World!”。实验中使用 module_init() 和 module_exit() 宏将这两个函数注册为模块的入口与出口函数，MODULE_LICENSE("GPL")、MODULE_AUTHOR()、MODULE_DESCRIPTION() 等宏用于描述模块的基本信息。通过编译生成 .ko 文件并使用 insmod、rmmod、dmesg 等命令验证运行结果。

Makefile：

```
obj-m += helloworld.o   # 指定模块文件

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules   # 编译模块
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean   # 清理编译文件
```

测试结果：

① 编译

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab$ make
make -C /lib/modules/6.17.014Jintao/build M=/home/tt/kernel_modules_lab modules
 # 编译模块
make[1]: 进入目录“/home/tt/下载/linux-6.17.2”
make[2]: 进入目录“/home/tt/kernel_modules_lab”
  CC [M]  helloworld.o
  MODPOST Module.symvers
  CC [M]  helloworld.mod.o
  CC [M]  .module-common.o
  LD [M]  helloworld.ko
  BTF [M] helloworld.ko
make[2]: 离开目录“/home/tt/kernel_modules_lab”
make[1]: 离开目录“/home/tt/下载/linux-6.17.2”
```

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab$ ls -l helloworld.ko
-rw-rw-r-- 1 tt tt 177048 10月  20 15:09 helloworld.ko
```

由上图可见编译成功。

② 装入模块

```
sudo insmod helloworld.ko
```

③ 查看日志

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab$ sudo dmesg | tail -n 10
[ 1436.865452] e1000: ens33 NIC Link is Down
[ 1443.011623] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
None
[ 1447.105272] e1000: ens33 NIC Link is Down
[ 1453.249994] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control:
None
[ 1483.382567] audit: type=1400 audit(1760943626.206:163): apparmor="DENIED" ope
ration="open" class="file" profile="snap.firmware-updater.firmware-notifier" nam
e="/proc/sys/vm/max_map_count" pid=6163 comm="firmware-notifi" requested_mask="r
" denied_mask="r" fsuid=1000 ouid=0
[ 2159.869031] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 4 times
, consider switching to WQ_UNBOUND
[ 2161.963363] workqueue: e1000_watchdog [e1000] hogged CPU for >10000us 5 times
, consider switching to WQ_UNBOUND
[ 2196.981561] helloworld: loading out-of-tree module taints kernel.
[ 2196.981586] helloworld: module verification failed: signature and/or required
 key missing - tainting kernel
[ 2196.998783] <1>Hello, World!
```

由上图可见，程序运行成功。

（2）/proc 目录操作

Procfs_example.c:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>
#include <linux/fs.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Procfs example module (safe version using single_open)");
MODULE_VERSION("1.0");

// 全局保存 /proc/procfs_example 目录指针
static struct proc_dir_entry *proc_dir;

// ---------- 1. 文件内容输出函数 ----------

// 输出 foo
static int foo_show(struct seq_file *m, void *v)
{
    seq_printf(m, "foo\n");
    return 0;
```

```
}

// 输出 bar
static int bar_show(struct seq_file *m, void *v)
{
    seq_printf(m, "bar\n");
    return 0;
}

// 输出 jiffies（系统启动以来的时间节拍数）
static int jiffies_show(struct seq_file *m, void *v)
{
    seq_printf(m, "jiffies: %lu\n", jiffies);
    return 0;
}

// ---------- 2. 打开函数，使用 single_open 简化 ----------

static int foo_open(struct inode *inode, struct file *file)
{
    return single_open(file, foo_show, NULL);
}

static int bar_open(struct inode *inode, struct file *file)
{
    return single_open(file, bar_show, NULL);
}

static int jiffies_open(struct inode *inode, struct file *file)
{
    return single_open(file, jiffies_show, NULL);
}

// ---------- 3. 定义文件操作结构体（proc_ops） ----------

static const struct proc_ops foo_ops = {
    .proc_open = foo_open,
    .proc_read_iter = seq_read_iter,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

static const struct proc_ops bar_ops = {
    .proc_open = bar_open,
    .proc_read_iter = seq_read_iter,
```

```
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

static const struct proc_ops jiffies_ops = {
    .proc_open = jiffies_open,
    .proc_read_iter = seq_read_iter,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

// ---------- 4. 模块加载函数 ----------

static int __init procfs_example_init(void)
{
    // 创建 /proc/procfs_example 目录
    proc_dir = proc_mkdir("procfs_example", NULL);
    if (!proc_dir)
    {
        printk(KERN_ALERT "Failed to create /proc/procfs_example directory\n");
        return -ENOMEM;
    }

    // 创建 foo 文件
    if (!proc_create("foo", 0444, proc_dir, &foo_ops))
        goto err;
    // 创建 bar 文件
    if (!proc_create("bar", 0444, proc_dir, &bar_ops))
        goto err;
    // 创建 jiffies 文件
    if (!proc_create("jiffies", 0444, proc_dir, &jiffies_ops))
        goto err;
    // 创建符号链接 jiffies_too -> jiffies
    if (!proc_symlink("jiffies_too", proc_dir, "jiffies"))
        goto err;

    printk(KERN_INFO "[procfs_example] Module loaded successfully\n");
    return 0;

err:
    // 若创建失败则清理已创建的文件
    remove_proc_entry("jiffies_too", proc_dir);
    remove_proc_entry("jiffies", proc_dir);
    remove_proc_entry("bar", proc_dir);
    remove_proc_entry("foo", proc_dir);
```

```
    remove_proc_entry("procfs_example", NULL);
    printk(KERN_ALERT "[procfs_example] Module load failed, cleaned up.\n");
    return -ENOMEM;
}

// ---------- 5. 模块卸载函数 ----------

static void __exit procfs_example_exit(void)
{
    if (proc_dir)
    {
        remove_proc_entry("jiffies_too", proc_dir);
        remove_proc_entry("jiffies", proc_dir);
        remove_proc_entry("bar", proc_dir);
        remove_proc_entry("foo", proc_dir);
        remove_proc_entry("procfs_example", NULL);
        proc_dir = NULL;
    }
    printk(KERN_INFO "[procfs_example] Module unloaded.\n");
}

module_init(procfs_example_init);
module_exit(procfs_example_exit);
```

编写了一个基于 `/proc` 文件系统的内核模块 `procfs_example`，通过创建虚拟文件展示内核与用户空间交互的机制。模块加载时在 `/proc` 目录下生成名为 `procfs_example` 的子目录，并在其中创建 `foo`、`bar`、`jiffies` 三个**只读（其中 jiffies 内容是动态变化的）**文件和一个指向 `jiffies` 的符号链接 `jiffies_too`。代码中分别定义了 `foo_show()`、`bar_show()` 和 `jiffies_show()` 三个输出函数，用于向用户输出固定字符串和系统启动以来的时钟节拍数 `jiffies`。在文件操作部分采用 `single_open()` 接口代替复杂的 `seq_operations`，从而简化了文件读取过程并避免空指针错误。模块加载函数 `procfs_example_init()` 负责创建目录和文件，卸载函数 `procfs_example_exit()` 负责释放资源并删除所有节点。

Makefile:

```
obj-m += procfs_example.o   # 指定模块文件
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules   # 编译模块
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean   # 清理编译文件
```

测试结果：

① 编译

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab/procfs_example$ make
make -C /lib/modules/6.17.014Jintao/build M=/home/tt/kernel_modules_lab/procfs_e
xample modules  # 编译模块
make[1]: 进入目录“/home/tt/下载/linux-6.17.2”
make[2]: 进入目录“/home/tt/kernel_modules_lab/procfs_example”
  CC [M]  procfs_example.o
  MODPOST Module.symvers
  CC [M]  procfs_example.mod.o
  CC [M]  .module-common.o
  LD [M]  procfs_example.ko
  BTF [M] procfs_example.ko
make[2]: 离开目录“/home/tt/kernel_modules_lab/procfs_example”
make[1]: 离开目录“/home/tt/下载/linux-6.17.2”
```

② 载入模块

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab/procfs_example$ ls /proc/proc
fs_example
bar  foo  jiffies  jiffies_too
```

③ 利用 CAT 查看

```
tt@tt-VMware-Virtual-Platform:~/kernel_modules_lab/procfs_example$ cat /proc/pro
cfs_example/foo
cat /proc/procfs_example/bar
cat /proc/procfs_example/jiffies
cat /proc/procfs_example/jiffies_too
foo
bar
jiffies: 4294839887
jiffies: 4294839890
```

由上图可知程序执行成功！

四.实验总结

（1）思考题：

1）说明为什么内核源代码中的输出函数选用了printk()而不是常用的printf()。

2）思考bar、foo、jiffies 和jiffies_too 文件分别是什么类型，它们是否可以进行读写。

3）总结并分析实验中出现的问题。

1. 内核态**没有标准 C 库**，不能链接/调用用户态的 printf()；而 printk() 是内核提供的日志接口，写入内核环形缓冲区（可被 dmesg/控制台读取），支持日志级别（如 KERN_INFO）、可在中断/原子上下文中使用、不会隐式休眠或使用浮点，适合早期启动与出错现场记录，因此内核源码必须用 printk()（或其封装 pr_info() 等）而不是 printf()。

2. bar、foo、jiffies、jiffies_too 都是由模块在 procfs（或 debugfs，依实验代码而定）创建的虚拟文件，并非磁盘常规文件；jiffies/jiffies_too 一般为只读，用于导出当前节拍计数 jiffies（每次读取都会变化，不支持写入）；foo/bar 通常是模块内的缓冲区映射成的条目，是否可读写**取决于创建时的权限位**（常见做法：foo 0644 可读、仅属主可写；bar 0200 仅可写，或 0644 可读写）。

3. 常见问题：内核版本/头文件不匹配导致编译或加载失败；误用 printf、缺少 MODULE_LICENSE 造

成 taint 提示；copy_to_user/copy_from_user 长度或返回值处理不当引发 -EFAULT/内核 oops；未按 proc_read/seq_file 语义处理 *ppos 导致 cat 死循环或只读一次就空；未在 module_exit 清理 proc/debugfs 条目造成“幽灵文件”；并发未加锁（spinlock/mutex）导致竞态；printk 日志级别过低/缓冲区被冲掉看不到输出；权限设置不合理导致读写被拒绝。排查时按“编译→加载(insmod/modprobe)→功能→卸载(rmmod)→清理”的链路逐点验证，并以 dmesg 为主定位。

（2）实验小结：

通过本次内核模块实验，我深入理解了 Linux 内核模块的基本结构与加载机制，并掌握了内核模块的编写、编译、加载与卸载的完整流程。首先，通过编写 helloworld 模块，了解了模块的基本组成部分，包括初始化函数、清理函数以及 MODULE_LICENSE、MODULE_DESCRIPTION 等宏的作用；在此过程中学习了 printk() 在内核日志中的输出机制。随后，通过实现基于 /proc 文件系统的 procfs_example 模块，掌握了如何在内核空间创建虚拟文件并实现与用户空间的交互，能够利用 single_open()、seq_printf() 等接口在 /proc 下动态输出内核数据，如 jiffies 的值。实验过程中还通过 Makefile 实现模块的独立编译与清理操作，并使用 insmod、rmmod、lsmod、dmesg 等命令验证模块的加载状态与运行结果。总体而言，本次实验不仅提升了我对 Linux 内核模块开发流程的理解，也加深了对内核与用户空间交互机制的认识，为后续更复杂的驱动开发和内核扩展奠定了基础。